Graphs

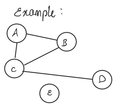
3/22/20 12:14 PM

Graph: Abstract data type representing connections.

* Very non-linear - more so than trees

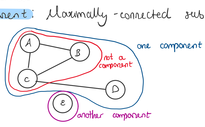
$G = (V, E)$ — graph; V = set of vertices; E = set of edges

Example:



Terminology:

- **Neighbor/adjacent** - occurs when nodes are connected by an edge.
- **Path**: Series of nodes connected by edges.
- **Cycle**: Path with same start and end node.
- **Connected**: A graph is connected if there is a path between all vertices.
- **Component**: Maximally-connected sub-graph.



Kinds of graphs:

- **Directed & Undirected**
  - The above graph is an example of an undirected graph.
  - A directed graph has directed edges called **arcs**:
  - $G = (V, A)$ where A is the set of arcs.
- **Cyclic vs. Acyclic**
  - Term: **DAG** = directed acyclic graph
- **Weighted vs. Unweighted**
  - Weighted graphs have values associated with the edges
- **Bipartite**
  - Is bipartite if the nodes can be partitioned into 2 sets such that each node is only connected to nodes in the other set.

→ Algorithm implementation below

Application	Node	Edge
- Social networks	People	Connection
- Transportation	Places	Roads
- Assignments/matching	Drivers/customers	Availability
- Web traffic	Websites	Citations

Traversal

Questions we might want to answer:

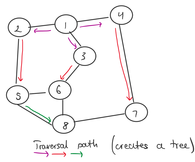
- Path from A to B?
- What are the components of a graph?
- Is there a cycle?

Methods:

Breadth-First-Search

Implementation:

```
def BFS(v):
    mark all nodes as unvisited
    q = Queue()
    while !q.empty():
        u = q.dequeue()
        mark u as visited
        q.enqueue(all unvisited neighbors of u)
```



(creates a tree)

Features:

- Will only search within component of initial node.
  - ↳ ∴ Can be used to find a component of a graph
- Can be used to find the shortest path (using unweighted edges) from any node to the starting node
  - ↳ Found via the traversal path

Complexity:

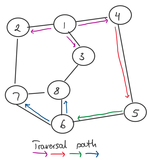
- Time: $O(\underbrace{\text{num. vertices}}_{n} + \underbrace{\text{num. edges}}_{m}) = O(n+m)$

  Justification: in the worst case, every node is enqueued and every edge is looked at twice. ?
- Space: $O(n)$

Depth First Search

Implementation:

```
def DFS(v):
    mark all nodes as unvisited
    s = Stack()
    s.push(v) and mark v as visited
    while !s.empty():
        u = s.pop()
        mark u as visited
        s.push(all unvisited neighbors of u)
```



Features:

- Can be implemented recursively (avoids use of stack)
- Can be used to find cycles

Complexity: Same as BFS

Worksheet notes

	1	2	3	4
1	✓	✓	X	✓
2	✓	✓	✓	X
3	X	✓	✓	X
4	✓	X	X	✓

1. $O(n)$ 2. $O(n)$ 3. $O(1)$ 4. $O(n)$

Can implement with a dict:

- key: node
- value: list of neighbor

```
def is_bipartite(v)
    mark all nodes as unvisited
    q = Queue(), X = set(), Y = set()
    q.enqueue(v) and mark v as visited
    X.add(v)
    while !q.empty():
        u = q.dequeue()
        mark u as visited
        u_set = u in X ? X : Y
        n = all neighbors of u
        for each node in n:
            if node in u_set:
                return False
            else:
                add "u" to opposite of u_set
        q.enqueue(unvisited neighbors of u)
    return True
```